**ISS4014 – Database Design**

**Chapter 10 – Row and Table Locking Lab**

**Introduction**

Locking mechanisms are critical to keep the database consistent (a fact that some developers do not understand well). However, unnecessary locks can have a negative impact on system performance. So, it’s crucial for any developer to understand the principles of locking and know when, where, and how to acquire a lock.

In this lab, we will introduce and explain the concepts of explicit locking mechanisms in *InnoDB* using several real-world examples. InnoDB is the database engine used in the MariaDB database within the XAMPP system.

Use the **Chapter 10 – Row and Table Locking Lab Response Sheet** to record the results of this lab as instructed. You will submit your Answer Sheet to Canvas with your responses recorded in the document at the completion of this lab to earn the points associated with this assignment.

**Locking Concepts**

There are basically two types of locking mechanisms:

* Read Lock (*Shared Lock*). These are locks taken on resources to provide consistent reads by not allowing write (updated, insert, or deletes) operations to that resource.

If no write lock is set on the resource, the transaction acquires the read lock immediately. Many transactions can acquire read locks on the same resource simultaneously. While a transaction holds a read lock, no write lock can be acquired on that resource, putting the transaction into the wait queue.

* Write Lock (*Exclusive Lock*). These are locks taken on a resource when it needs to be modified. There can be only one write lock on a resource at a given time. If there is no (read/write) lock on the resource, the transaction acquires the write lock immediately. While a transaction holds a write lock on a resource, all other lock requests must wait.

Notably, the write lock has a higher priority than the read lock. When a resource is unlocked, and if lock requests are waiting in the queue, then the lock is granted in the following manner:

* Grant the lock first to the request waiting in the write lock queue.
* If there is no lock request for the resource in the write lock queue, then grant the lock to the first request in the read lock queue.

**Locking in *InnoDB***

*InnoDB* supports both table-level and row-level locking.

Row-level locking offers good performance and throughput for heavy workloads where multiple transactions are reading from and/or writing to the same table simultaneously but writing to different rows without making each other wait.

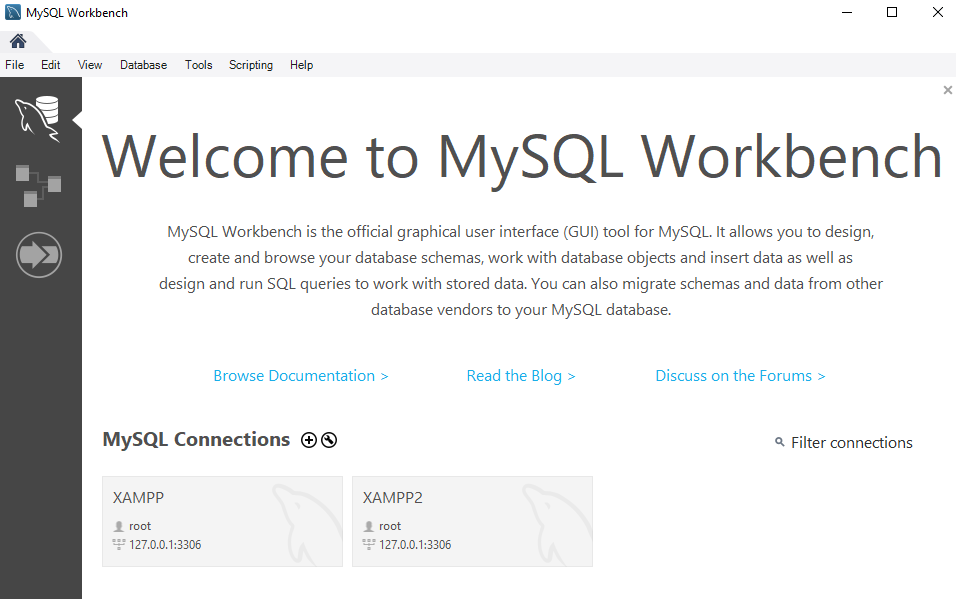
*InnoDB* also supports *Multi-Version Concurrency Control* (*MVCC*), an optimistic locking mechanism that allows different data snapshots to be available to different transactions. This mechanism implements the following transaction isolation levels: *Read Committed* and *Repeatable Reads*.

However, if the transaction isolation level is set to Serializable, the engine uses a pessimistic locking mechanism, traditionally implemented using the two-phase locking protocol.

**MySQL Workbench set up and Database Schema Creation.**

Let’s set up our MySQL Workbench to complete the lab. You should already have one connection to the XAMPP MariaDB defined on the MySQL Workbench home page. However, we must define a second connection to work through this lab.

**Create a second connection to the localhost XAMPP MariaDB server and name the connection XAMPP2. Once complete, your home screen should look like the following:**



To understand locking concepts, let’s create the *sample\_db* database and *testing\_tbl* table, as shown in listing 01. Open your XAMPP connection in MySQL Workbench and copy, paste, and run the following commands.

|  |
| --- |
| CREATE DATABASE sample\_db;  USE sample\_db;  CREATE TABLE sample\_db.testing\_tbl (   id int NOT NULL AUTO\_INCREMENT,   payload varchar(128) NOT NULL,   PRIMARY KEY (id) )ENGINE=innodb;  INSERT INTO sample\_db.testing\_tbl (payload) VALUES('row1');  INSERT INTO sample\_db.testing\_tbl (payload) VALUES('row2'); |

***Listing 01***

Refresh your database SCHEMAS listing and verify that the sample\_db schema has been created and the testing\_tbl is there with two records. You can do this by running a **select \* from testing\_tbl;**

**Row-level locking**

You can perform explicit row-level locking in *InnoDB* in two ways:

* *FOR UPDATE*
* *LOCK IN SHARE MODE*

Any lock held with the *FOR UPDATE* clause will not allow other transactions to read, update, or delete the rows until the transaction gets committed or rolled back, releasing the lock. This is basically an *exclusive/write lock*.

Let’s explain these concepts with examples. In your XAMPP connection session to the database, acquire an exclusive lock on a data row by running the commands as shown in listing 02.

|  |
| --- |
| # transaction 01 – XAMPP Session  # <your name> START TRANSACTION;  SELECT id, payload FROM sample\_db.testing\_tbl WHERE id=1 FOR UPDATE; |

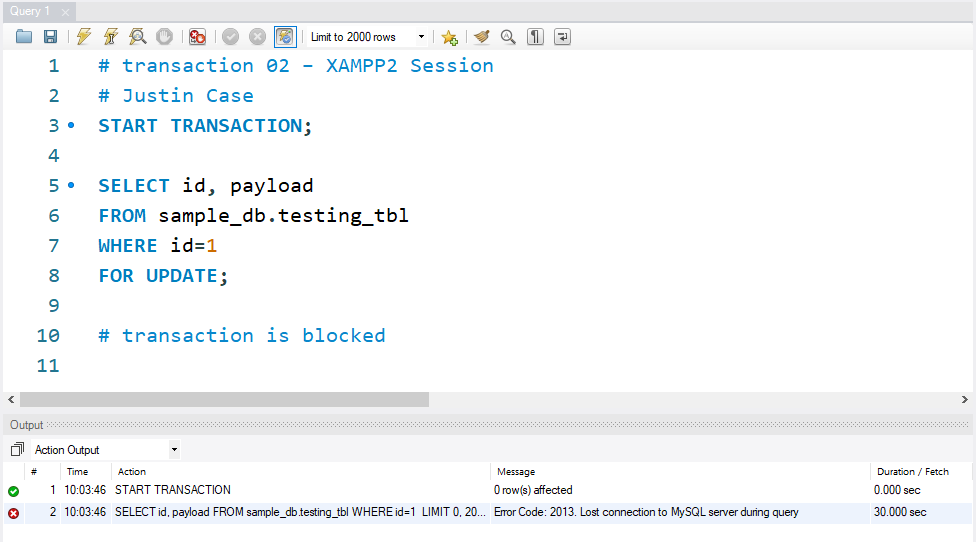
***Listing 02***

Now open your XAMPP2 connection to create a second session, and try to read (using the clause *FOR UPDATE*) the same data row from the table using the commands in Listing 03, and you’ll see that the transaction is blocked. After 30 seconds (or so), transaction 02 will time out.

|  |
| --- |
| # transaction 02 – XAMPP2 Session  # <your name>  USE sample\_db;  START TRANSACTION;  SELECT id, payload FROM sample\_db.testing\_tbl WHERE id=1 FOR UPDATE;  # transaction is blocked |

***Listing 03***

**Response 1** – Take an image of transaction 02 either running (waiting for the lock) or ended (Error Code: 2013), including the output log and paste the image in the answer sheet under the Response 1 heading. For example, the image should look like this:



Next, try to update or delete the same row from your XAMPP2 session, and you’ll see that the transactions will be put into the waiting state. Open a new SQL tab in your XAMPP 2 session and run the code in listing 04. You should see that the first transaction will first wait and then fail as it waits for the lock to be released that was set in Transaction 01 in the other XAMPP session. (No need to copy and paste a response for this activity).

|  |
| --- |
| # transaction 03 – XAMPP2 Session  # <your name>  START TRANSACTION;  UPDATE sample\_db.testing\_tbl  SET payload='row3'  WHERE id=1;  # transaction is blocked  ROLLBACK;  # transaction 04  START TRANSACTION;  DELETE FROM sample\_db.testing\_tbl  WHERE id=1;  # transaction is blocked  ROLLBACK; |

***Listing 04***

In order to release the lock, go back to the first XAMPP session and execute a COMMIT operation. Then, return to the XAMPP2 session. If you committed the transaction on the XAMPP session soon enough, you will see that transactions 03 and 04 ran when the lock was released. Otherwise, if the transaction failed due to time out, rerun transactions 03 and 04. The transactions should both now run. The ROLLBACK will undo the update and delete to keep the data in the same state.

**Response 2** – Take an image of transactions 03 and 04 from the XAMPP2 session after it completes, including the output log, and paste the image into the answer sheet under the Response 2 heading.

Any lock held with the *LOCK IN SHARE MODE* clause will allow other transactions to read the locked row but will not allow other transactions to write on the row until the transaction gets committed or rolled back and the lock is released. This is basically a *shared/read lock*.

Let’s explain with examples. Return to your XAMPP session and acquire an exclusive lock on a data row by modifying transaction 01 with a LOCK IN SAHRE MODE, as shown in listing 05.

|  |
| --- |
| # transaction 01 – XAMPP Session  # <your name> START TRANSACTION;  SELECT id, payload FROM sample\_db.testing\_tbl WHERE id=1 **LOCK IN SHARE MODE;** |

***Listing 05***

Now, return to your XAMPP2 session and the SQL tab with transaction 02. Update the SQL as shown in listing 06 and try to read (using the clause LOCK IN SHARE MODE) the same data row from the table, and you’ll see the transaction runs fine, but you can’t update or delete the data row.

|  |
| --- |
| # transaction 02 – XAMPP2 Session  # <your name> START TRANSACTION;   SELECT id, payload FROM sample\_db.testing\_tbl WHERE id=1 **LOCK IN SHARE MODE;** # transaction runs fine   UPDATE sample\_db.testing\_tbl SET payload='row3' WHERE id=1; # transaction is blocked  ROLLBACK; |

***Listing 06***

**Response 3** – Take an image of transaction 02 from the XAMPP2 session after it completes (either waiting or times out), including the output log, and paste the image in the answer sheet under the Response 3 heading.

In order for transaction 02 to keep on running, we need to go back to transaction 01 and commit the changes, releasing the lock.

Return to the XAMPP session and run a COMMIT operation to release the lock. Return to the XAMPP2 session and rerun the transaction 02 script. It should run both the select and the update statements and complete with a rollback.

Now, let’s complete one more activity to demonstrate the row level locking. Go back to the XAMPP session and run transaction 01 (excluding a COMMIT operation to put a lock on the row 1 record). Return to the XAMPP 2 session and modify transaction 02 so the WHERE clause is set to id=2, as shown in listing 07. Run the transaction.

|  |
| --- |
| # transaction 02 – XAMPP2 Session  # <your name> START TRANSACTION;   SELECT id, payload FROM sample\_db.testing\_tbl **WHERE id=2** LOCK IN SHARE MODE; # transaction runs fine   UPDATE sample\_db.testing\_tbl SET payload='row3' **WHERE id=2;** # transaction is blocked  ROLLBACK; |

***Listing 07***

You should see that the transition is complete with no issues. This was because in the XAMPP session, the lock was only on record row 1, but in the XAMPP2 session, the update was on record row 2.

**Response 4** – Take an image of transaction 02 from the XAMPP2 session after it successfully completes, including the output log, and paste the image in the answer sheet under the Response 4 heading.

**Table-level locking**

In *InnoDB*, transactions can also acquire locks at the table level.

In order to acquire a table lock, we can use the following statement: *LOCK TABLES* table name *[READ | WRITE]*.

In order to release a table lock, we must execute the following statement: *UNLOCK TABLES*.

*READ/WRITE* locking mode is similar to the ones explained above.

Let’s test each locking mode.

*WRITE TABLE LOCK.*

Notice that only the transaction that holds the *WRITE* table lock can read and write data from the table.  Other transactions cannot be read and written from the table until the *WRITE* table lock is released.

Go to the XAMPP session and execute the transaction in listing 08 that acquires a *WRITE* table lock. We can insert a new row into the table or read data inside this transaction.

|  |
| --- |
| # transaction 01 – Table Lock – XAMPP Session  # <your name> START TRANSACTION;   LOCK TABLE sample\_db.testing\_tbl WRITE;   SELECT  id, payload FROM sample\_db.testing\_tbl; # transaction runs ok   INSERT INTO sample\_db.testing\_tbl (payload) VALUES('row3'); # transaction runs ok |

***Listing 08***

**Response 5** – Take an image of transaction 01 from the XAMPP session after it successfully completes, including the output log, and paste the image into the answer sheet under the Response 5 heading.

Go to the XAMPP2 session, create the transaction shown in listing 09, and try to read data from the table or insert a new row, and we can see both operations are put into a waiting state.

|  |
| --- |
| # transaction 02  # <your name> START TRANSACTION;   SELECT  id, payload FROM sample\_db.testing\_tbl; # transaction is blocked   # transaction 03 START TRANSACTION;   INSERT INTO sample\_db.testing\_tbl (payload) VALUES('row4'); # transaction is blocked  ROLLBACK; |

***Listing 09***

**Response 6** – Take an image of transaction 02 from the XAMPPS session after it starts (while it is running or after it times out), including the output log, and paste the image in the answer sheet under the Response 6 heading.

In order for the first transaction to release the table lock, we can execute the statement shown in listing 10.

|  |
| --- |
| # transaction 01 UNLOCK TABLES; |

***Listing 10***

Run transaction 02 in the XAMPP2 session to ensure it completes successfully now that the table lock has been released.

*READ TABLE LOCK*.

Go to the XAMPP session and modify transaction 01 to what is shown in listing 11, and execute the transaction to acquire a read table lock.

|  |
| --- |
| # transaction 01 START TRANSACTION;   LOCK TABLE sample\_db.testing\_tbl READ; |

***Listing 11***

Go to the XAMPP2 session and execute transaction 02 again (see listing 12). Note that the select statement will execute, but the insert statement is put into a waiting state due to the table read lock.

|  |
| --- |
| # transaction 02  # <your name> START TRANSACTION;   SELECT  id, payload FROM sample\_db.testing\_tbl; # transaction is blocked   # transaction 03 START TRANSACTION;   INSERT INTO sample\_db.testing\_tbl (payload) VALUES('row4'); # transaction is blocked  ROLLBACK; |

***Listing 12***

**Response 7** – Take an image of transaction 02 from the XAMPP2 session after it starts (while it is running or after it times out) showing the completion of the select statement and the waiting on the insert statement, including the output log and paste the image in the answer sheet under the Response 6 heading.

In order for the transaction in your XAMPP session to release the table lock, we can execute the statement shown in Listing 13 in your XAMPP session.

|  |
| --- |
| # transaction 01 UNLOCK TABLES; |

***Listing 13***

Run transaction 02 in the XAMPP2 session to ensure it completes successfully now that the table lock has been released.

Acknowledgements:

Lab derived from Juan Carlos Olamendy work on his Blog at https://blog.toadworld.com/2018/01/11/explaining-innodb-explicit-locking-mechanisms